Abstract

Incited by the observation that cluster analysis and the remodularization of software systems solve similar problems, we have done research in both these areas in order to provide theoretical background for the application of cluster analysis in systems remodularization. In this article we present an overview of cluster analysis and of systems remodularization. It appears that system remodularization techniques often either reinvent clustering techniques or could be augmented by them. We also give directions for further research.

note: The author was sponsored by bank ABN AMRO, software house DPFinance, and the Dutch Ministry of Economical Affairs via the Senter Project #ITU95017 “SOS Resolver”.

1. Introduction

While organizations demand changes to their legacy systems at an ever accelerating rate to serve the needs of their (core) business, these systems become less and less flexible because of these changes. System evolution over a decade or longer inevitably results in a very interwoven system structure as making resources available to prevent this would result in enormous backlogs. Due to this structure minor changes in some part of the system may have unforeseen effects on other parts of the system which may not function correctly anymore without being changed themselves. These misty dependencies seriously harden maintenance and addition of extra functionality. According to Lehman [17], systems evolution will eventually result in the downfall of the system unless serious effort is put into ameliorating the systems structure.

Rewriting the entire system is often not a feasible solution. (The reasons for this lie beyond the scope of this article, one of them concerns the risk and the price of redoing a project which took over a decade). An approach which seems more feasible is to try to reduce the complexity of the system’s structure.

An often used way of reducing complexity in software engineering is to chop up a system in chunks, called modules or units, which inner structure can be changed without affecting the other chunks. This is done by explicitly defining the interrelationships between modules through interfaces. These interfaces are kept as simple as possible to achieve a low degree of coupling between the modules. Once a system has been divided in relatively independent subsystems, it becomes feasible to replace an entire subsystem without seriously affecting the rest of the system.

In normal (forward) software engineering practice the modules and their interrelationships are defined before any programming takes place. In the case of legacy systems however, the program structure is already there. To achieve a system consisting of relatively independent modules we must remodularize the legacy system, i.e. reverse engineer the chunks.

We can view remodularization in two ways depending on the level of granularity we choose. At the system level remodularization means splitting up the system to form smaller chunks. At the entity level remodularization means grouping the entities to form larger chunks. An important question is how a good split or grouping can be achieved. In many scientific areas similar questions have been raised. In these areas research has been done on so called clustering or partitioning methods. As clustering methods are concerned with grouping entities based on their interrelationships or similarity, legacy systems renovation could benefit from the achievements made in this research area.

The goal of this article is to provide an overview of clustering methods. This overview can serve as input for remodularization projects and conversion scenarios towards an object oriented architecture. An overview of the literature in these fields is provided as well. The approaches taken have sometimes been influenced by clustering methods. Other ones could very well be improved by incorporating ideas.
from clustering methods. We feel that future research can benefit from the synergy which will result from the combination of the two fields.

This research is part of the larger project “Resolver” which is conducted by the Centre of Mathematics and Computer Science (CWI), the University of Amsterdam and ID Research. The goal of the Resolver project is to define and implement a generic ‘system renovation factory’ based on scientific research.

The outline of this article is as follows: In sections 2 to 4 we discuss general cluster analysis. Approaches to systems remodularization from the literature are presented in section 5. Finally we present our conclusions and directions for further research in section 6.

2. Introduction to Clustering

Partitioning or clustering techniques are used in many different areas for a wide spectrum of problems. Among the areas in which cluster analysis is used are graph theory, business area analysis, information architecture, information retrieval, resource allocation, image processing, software testing, galaxy studies, chip design, pattern recognition, economics, statistics, and biology. A major stimulus to the field of clustering was given by [29]. Their work is in the field of ‘numerical taxonomy’ which is concerned with the classification of species by investigating data on a set of organisms and grouping similar organism together.

In abstract terms we can formulate modularization as the grouping of large amounts of things in groups (modules) in such way that the things in one group are closely related compared to the relationships between things in different groups. In cluster analysis such groups are called clusters. [7] defines clusters in a similar way as “continuous regions of space containing a relatively high density of points, separated from other such regions by regions containing a relatively low density of points”. This is a very general definition which appeals to our intuition. Figure 1 illustrates the notion of clusters in terms of spatial density. The items which are to be clustered (the points) are represented by bullets. The dotted shapes represent clusters, the points within each dotted shape constitute a cluster.

We have so far used the terms ‘thing’, ‘point’ and ‘item’ to denote the raw material that is to be clustered. [29] introduce the ‘OTU’ (operational taxonomic unit). [2] gives as other possible terms ‘data unit’, ‘subject’, ‘observation’, ‘case’, ‘element’, ‘object’ or ‘event’. [1] further mentions the use of the terms ‘entity’ and ‘pattern’. Most of these terms are extremely overloaded in computing science. To avoid a terminologic chaos, we will only use the term ‘entity’ when we refer to this raw material.

The goal of clustering methods is to extract an existing ‘natural’ cluster structure. However, different methods may come up with different clusterings. So a particular algorithm may impose a structure rather than find an existing one. It might even be the case that an algorithm ‘finds’ a structure while there really is no natural structure in the data. Random hypotheses (performing the algorithm on random data sets which have no structure) can be used to check on this phenomenon. When the entities are pieces of software (systems) imposing a structure need not be a problem. It can actually be turned into an advantage as imposing a structure on those pieces is exactly what we want to achieve. By selecting an appropriate method we can steer the clustering and make it suit our ideas of a good modularization.

When applying cluster analysis a number of questions need to be answered:

- What are the entities to be clustered?
- When are two entities to be found similar?
- What algorithm do we apply?

In the case of software systems possible entities are chunks of data or procedural chunks, in section 5 several existing methods and their choice of entities are discussed. In the next section we discuss measures of similarity which are used to answer the second question. A categorized overview of clustering algorithms is presented in section 4.

3. Similarity

One question which obviously comes to mind when reading the definitions of clusters is how to determine whether two entities are closely related or in other words, whether a region contains a relatively high density of entities. In the literature on clustering this question is often phrased in terms of similarity ([1] also mentions the use of synonyms like ‘resemblance’, ‘proximity’ and ‘association’). Cluster algorithms group similar entities together. In order to talk
about the similarity of entities and say things like “entity \(a\) is more similar to entity \(b\) than it is to entity \(c\)” we need some kind of measure of similarity. The similarity measures in the literature can be divided in two groups. These two groups are characterized by the kind of information on entities which serves as input for the computation of the similarity measures:

- relationships between the entities
- for each entity it’s scores for a number of variables

The first sort can be translated in a similarity measure quite straightforwardly. When we think of the entities as nodes in a graph and the relations between them as edges in that graph, it suits our intuition to say that the more edges (relations) there are between two entities, the more similar they are. The edges can be weighted in which case the sum of the weights of the edges are taken and the edges can be directed or undirected. We could also distinguish between different kinds (colors) of edges.

The second sort is often found in statistics: the entities are scored on a number of attributes or variables. Persons are questioned about their favorite color, age, education and so on. These aspects are used to determine the similarity between the entities (persons) by serving as input for the similarity measures. Again we face a divers use of terminology and again most of the terms used, are overloaded in computing science. [1] mentions the terms ‘variable’, ‘attribute’, ‘character’ and ‘feature’ to represent aspects of entities. In this article we will use the term ‘feature’ as this is probably the least overloaded term.

Features can be divided into two groups determined by the scales used for the scores [2]:

- categorical or qualitative features, which are scored on either a nominal or an ordinal scale
- quantitative features, which are scored on either an interval scale or a ratio scale

A special variant of nominal features are binary or dichotomous features which can take only two values which are usually represented by 0 and 1. Binary features are often subdivided in symmetric and asymmetric. For symmetric binary features the 0 and 1 are just labels and are interchangeable. For asymmetric binary features 1 often means that the feature is present and 0 indicates its absence [13]. As a result 0 and 1 can not always be interchanged as two entities which both score 1 (feature present) are often more alike than two entities which both score 0.

In practice several features are used to cluster the entities. It is possible that not all of these features are measured on the same scale. Discussions on some of the problems and possible workarounds in the area of mixed features can be found in [2] and [13].

### 3.1. Similarity measures

There are a lot of different similarity measures which compute the similarity between entities based on the scores on selected features. A similarity measure always yields a value between 0 and 1. Two entities are more similar when their similarity measure comes closer to 1. Often dissimilarity measures are used. From these measures, similarity measures can easily be computed as follows: \(\text{sim}(i, j) = 1 - \text{dis}(i, j)\). [29] gives the following categories of similarity measures: distance measures, association coefficients, correlation coefficients and probabilistic similarity measures. We will briefly describe these measures. For a more elaborate discussion refer to [29].

#### Distance measures

Distance measures measure the dissimilarity of entities. The greater the outcome the more dissimilar the entities are. The distance between two entities should be zero if and only if the entities have the same scores on all features. The most popular distance measures are the (squared) Euclidean distance and the Manhattan distance or city-block metric. Both measures are special cases of the Minowski metrics (see e.g. [7]).

#### Association coefficients

Association coefficients or matching coefficients for two entities \(i\) and \(j\) are expressed in terms of the number of features which are present for these entities. So association coefficients assume binary features. In the literature the following table is used for this purpose (e.g. [29]):

<table>
<thead>
<tr>
<th>Entity (i)</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(a)</td>
<td>(b)</td>
</tr>
<tr>
<td>1</td>
<td>(c)</td>
<td>(d)</td>
</tr>
</tbody>
</table>

\[ a + b + c + d \]

In this table \(a\) represents the number of features for which both entities have the value 1, \(b\) represents the number of features present for entity \(i\) but absent for entity \(j\) etc. The idea behind association metrics is very intuitive: the more relevant matches there are between the sets of present features of the two entities under comparison, the more similar the two entities are.

Several choices are possible as to what features are relevant and more sophisticated coefficients use weighting. In the literature a lot of different association coefficients are presented. This diversity is caused by two factors [2]:

- The handling of 0-0 matches. Should 0-0 matches positively influence the similarity measure and if so should
they contribute with the same strength as 1-1 matches?

- The weighting of matches and mismatches.

[2] presents a list of possible coefficients by mechanically applying combinations of possible choices for these two factors. Some of the derived coefficients are widely used and are known under a specific name, others have no sensible meaning however. In this article we will only discuss the most frequently used coefficients. The simple matching coefficient is defined as:

$$\text{simple}(i, j) = \frac{a + d}{n}$$

This coefficient treats 1-1 matches and 0-0 matches equally, both contribute to the similarity. Matches and mismatches are weighted equally. The Russel and Rao coefficient excludes the 0-0 matches. The Jaccard coefficient doesn’t take 0-0 matches into account at all:

$$\text{Jaccard}(i, j) = \frac{a}{a + b + c}$$

Therefore this measure is very well suited for asymmetric binary features. The difference in weighting between matches and mismatches is usually done by a factor 2. The Dice coefficient applies double weighting of 1-1 matches to the Jaccard coefficient. A lot of other weightings are possible and are presented in the literature (see e.g. [2] and [13]).

**Correlation coefficients**

Correlation coefficients are originally used to correlate features. They are applied to the correlation of entities as well although it makes no statistical sense to obtain the mean value across different feature types rather than across entities [1]. The most popular coefficient of this sort is the Pearson product-moment correlation coefficient. The value of a correlation coefficient lies in the range from -1 to 1. A value of 0 means that the two entities are not related at all.

In [13] two formulae are presented to convert correlation coefficients to dissimilarity measures. Both formulae treat entities with a high positive correlation as very similar. The difference lies in the way negative correlations are handled.

**Probabilistic measures**

Probabilistic measures [1, 29] are based on the idea that agreement on rare features contributes more to the similarity between two entities than agreement on features which are frequently present. So probabilistic coefficients take into account the distribution of the frequencies of the features present over the set of entities. When this distribution is known, for each feature a measure of information or entropy can be computed.

The entropy quantifies the disorder, variance, confusion or surprisal. The two (sets of) entities which provide the least information gain (change of entropy) when combined have the highest similarity. For a more detailed discussion on probabilistic coefficients we refer to [29].

### 4. Clustering algorithms

In section 2 a lot of application areas of modularization were mentioned. The techniques used in these different areas can roughly be divided in the following categories:

- graph theoretical algorithms
- construction algorithms
- optimization algorithms
- hierarchical algorithms

Some authors (like [1]) present more categories which contain very specific algorithms or variants of other algorithms. Most algorithms presented in individual articles and developed to solve a specific problem in a specific area do not fit in any category at all. They are hybrid methods using a melange of useful ideas from different categories which proved to be a good combination for the problem at hand.

The reason for presenting a categorization of algorithms here is to give an overview of what’s in store. The next step will be to pick the right ingredients from the different categories to cook algorithms which are suited for the remodularization of legacy systems and the classification of their components.

Most algorithms produce disjoint clusters. There are methods which allow clusters to overlap and other methods may be adapted to allow this as well. In most usages of clustering methods people do not want clusters to overlap however. [23] suggests a compromise in which overlapping clusters are generated which are then ‘tidied up’ to yield disjoint clusters.

Cluster algorithms can be either supervised or unsupervised. An unsupervised algorithm is not provided any a priori knowledge, a supervised algorithm is. Such a priori knowledge can be the number of clusters for example. Most algorithms have more or less supervised variants. Some algorithms even allow interactive supervision.

We will now discuss each category and give a general description of the algorithms used.

#### 4.1. Graph theoretical algorithms

Graph theoretic algorithms work on graphs. The nodes of such graphs represent entities and the edges represent relations. Graph algorithms do not start from the individual nodes (entities), but try to find subgraphs which will form
the clusters. Special kinds of subgraphs like connected components, maximal complete subgraphs or spanning trees are used to derive modules or are candidates themselves. The algorithms used to find these special subgraphs are provided by or based on graph theory. Often applied examples of algorithms which fit in this category are minimal spanning tree clustering [10, 23, 25, 24] and aggregation algorithms.

### Aggregation algorithms

Aggregation algorithms reduce the number of nodes (representing entities) in a graph by merging them into aggregates. The aggregates can be used as clusters or can be the input for a new iteration resulting in higher level aggregates.

The graph reduction technique [31] selects nodes (one at a time) and makes a new aggregate-node containing the selected node together with its neighborhood set (the set of nodes no further than \(r\) edges away). For each node \(r\) is determined so that the resulting aggregate-node will contain \(R\) nodes. \(R\) is the degree of reduction. Less supervised variants allow variable values for \(R\).

In [4] a technique is described to find aggregate-nodes based on the graph theoretic notion of \(k\)-components. The algorithm first takes out the strongest \(k\)-components (the largest \(k\)) and proceeds with lower values for \(k\). The author defines the frontier of an aggregate as the set of nodes in this aggregate that have links to other aggregates (compare this with the notion of interface!).

### 4.2. Construction algorithms

The algorithms in this category assign the entities to clusters in one pass. The clusters may be predefined (supervised) or constructed as part of the assignment process (unsupervised).

We use the term geographic techniques for those techniques which use information like the location of entities in a two dimensional plane. An example is the bissection algorithm presented in [31]. This algorithm divides the plain in two. Entities which lie on the same part belong to the same cluster.

Density search techniques follow the definition of clusters given in section 2. They explicitly try to find regions containing a relatively high density of points. These regions of high density are also called modes. The dense entities initiate clusters. Clusters can be merged to form new clusters and individual entities can be assigned to clusters. The algorithms gradually move to a final clustering with an explicit stop criterion. However, no entities are relocated such as is the case with optimization algorithms.

Mode analysis [34] finds ‘dense entities’ by computing for each entity the number of entities which are contained in a sphere of some radius \(R\) surrounding that entity. An entity is dense if this number is greater than some value \(K\). In [9] an algorithm is presented which is based on fuzzy sets. An ordering is defined on entities determined by their grade of membership (defined by the characteristic function of the fuzzy set). Following this order, each entity is either assigned to the last initiated cluster or it is used to initiate a new cluster, depending on the distance to the entity which was used to initiate the last initiated cluster.

### 4.3. Optimization algorithms

An optimization or improvement algorithm takes an initial partitioning and tries to improve this solution by iterative adaptations according to some heuristic. So optimization algorithms are evolutionary algorithms.

The most discussed optimization methods are called ‘partitioning techniques’ [7] or simply ‘non-hierarchical clustering methods’ [2]. We will use the term ‘iterative partitioning’. Iterative partitioning algorithms start with an initial partition (clustering) in which entities are moved to other clusters in order to improve the partition according to some criterion. This relocating goes on until no further improvement of this criterion takes place. [1] speaks of ‘hill climbing passes’ in this respect.

The most widely used iterative partitioning methods make use of ‘seed points’ and fit the following scheme:

1. find an initial partition of \(k\) clusters
2. repeat determine the seed point of each cluster
   move each entity to the cluster with the most similar seed point
   until no entities were relocated in this iteration

In [2] it is proven that convergence indeed takes place. For the seed point of a cluster usually the centroid is taken. The centroid of a cluster is the ‘centre point’ of a cluster, the entity which most resembles ‘the average entity’ scoring the average value for each feature. Other parameters of this scheme are the similarity measure used (see section 3.1) and the way an initial partitioning is obtained.

The initial partition can be a clustering resulting from another type of clustering algorithm, like a hierarchical algorithm. Other possibilities for the initial partition are a random partition or a partition which grows by seeding \(k\) initial seed points which will each form a cluster and assigning each entity to the cluster with the most similar seed point (as such performing a first iteration). The more sophisticated methods assure that the seed points are rather dissimilar or even span the set of entities or they let the user pick the seed points giving the latter the possibility to influence the clustering by favoring a particular aspect while seeding. [2] and [7] provide some references on this matter. It is not clear whether an unlucky choice for the initial partition will cause the al-
algorithm to get stuck in a local optimum or if eventually a global optimum will always be reached.

An often used variant of the above scheme is the $k$-means method [19]. In this method the centroids (centre points) of the involved clusters are recomputed after each relocation of an entity. Only one such pass is performed. Now the resulting centroids remain fixed and the other entities are reallocated in one last iteration like was done in the general scheme. This algorithm is reported to be very fast and applicable to large data sets but its result depends on the order in which the entities are relocated.

In the above scheme the number of clusters $k$ was fixed. Several adaptations of the scheme have been developed which allow the number of clusters to vary during the process (e.g. [19]).

Another variant is provided by the ISODATA method, originally presented in [3] and described in e.g. [2] and [13]. This method has 7 (!) control parameters constraining the maximum number of iteration passes, the minimum number of entities in a cluster (if there are less, the cluster is discarded) and the minimum and maximum number of clusters. Furthermore there are so called lumping and splitting iterations in which clusters are merged or split respectively. When such iterations are performed again depends on parameters.

Clumping Techniques
Another form of algorithms which perform optimization are so called clumping techniques. In each iteration one cluster is identified. (see e.g. [7]). By repeated iteration different clusters (or clumps) are found which may overlap. A negative aspect of this method is that finding the same clump several times can not be avoided completely.

4.4. Hierarchical algorithms

There are two kinds of hierarchical algorithms: agglomerative and divisive algorithms. Both build a hierarchy of clusterings in such way that each level contains the same clusters as the first lower level except for two clusters which are joined to form one cluster. Figure 2 shows an example of such a hierarchy for three entities.

Agglomerative algorithms start at the bottom of the hierarchy: at the starting point there are $N$ clusters each containing one entity ($N$ is the number of entities). In each following step two clusters are joined. After $N - 1$ steps all entities are contained in one cluster. Each level in the hierarchy defines a clustering. Now a cut point has to be determined. The clustering at the level of the cut point is the resulting clustering. Divisive clustering works the other way around. At the beginning all entities are contained in one cluster. In each step a cluster is split into two clusters. After $N - 1$ steps there are $N$ clusters each containing one entity.

The resulting hierarchy of a hierarchical method is usually visualized in a dendrogram. In figure 3 a dendrogram of the example in figure 2 is shown. The entities are represented by numbers in order of appearance from left to right. The dotted line marks a possible cut point. In this example, cutting the hierarchy at the cut point would result in a cluster containing the two leftmost entities and a cluster containing only the rightmost entity (this is the clustering at the second level of the hierarchy). According to [13] divisive algorithms offer an advantage over agglomerative algorithms because most users are interested in the main structure of the data which consists of a few large clusters found in the first steps of divisive algorithms. Agglomerative algorithms start with the details (the individual entities) and work their way up to large clusters which may be affected by unfortunate decisions in the first steps. Agglomerative hierarchical algorithms are most widely used however. This is because it is infeasible to consider all possible divisions of the first large clusters ($2^{N-1} - 1$ possibilities in the first step). So there are
not many feasible divisive algorithms but there are divisive algorithms for which there exists an agglomerative counterpart that produces the same hierarchy [30].

We will now discuss agglomerative and divisive methods in more detail.

### 4.4.1. Agglomerative methods

Agglomerative hierarchical methods all fit the following scheme, known as Johnson’s algorithm (adapted from [12]):

1. begin with \( N \) clusters each containing one entity (\( N \) is the number of entities) and compute the similarities between the entities (clusters)

2. while there is more than 1 cluster
   - find the most similar pair of clusters
   - merge these clusters into a single cluster
   - update the similarities between the clusters

Often algorithms are presented in terms of dissimilarity. In this case the two clusters which are the least dissimilar are joined. The different algorithms all follow the scheme above, they use different parameters however. The parameters which follow from the scheme are:

- the similarity measure
- the updating rule

Similarity measures were treated in section 3.1 In the following paragraph we will discuss the updating of the similarities.

### Updating similarities

When two clusters are joined or one cluster is split up in two smaller clusters, the similarities between the newly formed cluster(s) and the previously existing ones have to be calculated. When a similarity measure was used that can be applied to sets of entities these similarities can be computed from the original data. Otherwise the computation is done by means of an updating rule. An updating rule takes the similarities (and possibly some additional information like the number of entities in each cluster) from the previous step of the algorithm as input and computes a value which serves as the new similarity measure.

The following discussion of updating rules assumes that two clusters, \( A \) and \( B \), have been joined. The similarity between an already existing cluster \( C \) and this new cluster is computed.

#### Single and complete linkage

An often used updating rule is the single linkage rule or nearest neighbor updating rule:

\[
\text{single link}(C, A \cup B) = \max \left( \text{sim}(C, A), \text{sim}(C, B) \right)
\]

So cluster \( C \) is as similar to \( A \cup B \) as it is to the most similar of the old clusters.

The complete linkage rule or furthest neighbor rule takes the similarity with the least similar old cluster to be the new similarity:

\[
\text{compl link}(C, A \cup B) = \min \left( \text{sim}(C, A), \text{sim}(C, B) \right)
\]

The resulting clustering depends on the updating rule used. When we apply different updating rules on the same data, we might find different clusterings. As an example consider figure 4 where the single linkage and complete linkage rule were used on the same data. The single linkage rule will prefer non-compact and isolated clusters whereas complete linkage will find compact clusters which may not be very isolated.

![Figure 4. different clusterings produced with different update rules: single linkage and complete linkage respectively](image)

In figure 4 we also see that single entities can have a great influence on the resulting clustering. If, in case the single linkage update rule is used, two ‘end-points’ of two clusters are close to each other, the clusters may end up as one although some entities may be very far apart from each other. This effect is known as ‘chaining’ as chains of entities can be created. This is indeed the case in the leftmost case in figure 4. On the other hand, when applying the complete linkage update rule, the opposite situation occurs. In order for two clusters to be merged, each entity in one of the two clusters should be very close to every other entity in those clusters (because even the two entities which are the furthest apart should be close to each other) so outlying points may prevent the merger (in this step of the algorithm). So complete linkage may result in compact clusters which are not very well separated. In the rightmost clustering in figure 4 we see indeed that the rightmost entity of the cluster on the left and the leftmost entity of the cluster on the right are rather close to each other.

[13] say single linkage to be space contracting as rather dissimilar entities can be grouped together whereas com-
complete linkage is space dilating because relatively similar entities are kept apart for a long time (this is also called the dissection effect). To find a compromise between these two extreme methods, several space conserving methods have been developed.

Other update rules
An approach which comes to mind when compromising between the extremes presented above is to take the average of the original similarities to be the new similarity. There are however different ways to determine ‘the’ average similarity producing different results. The simplest form is weighted average linkage or weighted pair group average method (often abbreviated to WPGMA):

$$\text{wpgma}(C, A \cup B) = \frac{1}{2} \text{sim}(C, A) + \frac{1}{2} \text{sim}(C, B)$$

Although there are no weights in the formula, implicit weighting takes place. If one of the two clusters, say $A$, is larger (contains more entities) than the other, the entities of cluster $B$ will carry a larger weight. Furthermore this rule may produce different results when clusters with the same similarity are joined in a different order [13].

These problems are overcome by the unweighted average linkage rule, unweighted pair-group average method (often abbreviated to UPGMA) or the group average rule which assigns the same weight to all pairs of entities.

For interval-scaled features the centroid method can also be used. This method uses the similarity (in terms of the Euclidean distance) between the centroids (centre points) of the clusters. Like the average methods, the centroid method also comes in an unweighted (UPGMC) and a weighted version (WPGMC). The weighted version is also known as Gower’s method or the median method. More details on centroid clustering and the exact (rather complicated) update rules involved can be found in [13].

Another method which is often found in the literature is Ward’s method [32]. This method tries to optimize the minimum variance within clusters by ensuring that those clusters are joined that cause a minimal increase in the within-groups sum of squares or error sum of squares. A derivation of an update rule for Ward’s method can be found in [13].

In [15] a general formula is presented which can be used to describe update rules. The formula has 4 parameters. Filling in the right values for these parameters will yield the update rules discussed here.

4.4.2. Divisive methods
As we noted earlier the divisive hierarchical approaches are burdened by a huge computational complexity. To cope with this complexity, methods have been devised which consider only a subset of all possible divisions. Feasible divisive hierarchical methods can be either monothetic or polythetic.

Monothetic methods are mostly used with binary features. The division of a cluster is determined by certain features (usually one) on which certain scores are necessary to belong to a certain new cluster. The best known variant of monothetic divisive clustering is association analysis. In this method only one feature is used for the splitting. This results in a cluster in which all entities possess that feature and a cluster in which no entity possesses it. The splitting feature is chosen in such way that the similarity between the newly formed clusters is minimal in terms of a certain criterion (e.g. information loss [16] which should be maximized because it is a dissimilarity measure). In the next step of the algorithm, another feature is selected for the splitting of the clusters. This need not be the same feature for all clusters. Following this procedure the resulting hierarchy is equivalent to a decision tree in which each node is labeled with the feature used for splitting. For a little more elaborate discussion on monothetic divisive methods see [7], [13] or [10].

In polythetic methods the possession of a certain subset of the features suffices for an entity to belong to a cluster, no features are compulsory. Other definitions say that in polythetic methods all features are taken into account (e.g. to compute a similarity measure) whereas monothetic methods only look at one feature at every level.

In [18] dissimilarity analysis, which is one of the most feasible polythetic methods, is discussed. In this method a cluster $A$ is split by taking out the entity $a$ for which $\text{sim}(a, A - \{a\})$ is minimal (the original description by [18] was in terms of dissimilarity). For this computation several similarity measures working on an entity and a cluster can be used. In [7] and [13] the average Euclidean distance is used. The entity $a$ is used to form a new cluster, called splinter group by [7] and [13]. Now a number of iterations is performed. In each iteration, that entity which is the ‘most more similar’ to $S$ than to $A$ is moved to $S$ and the similarities are recomputed. The resulting clusters $A$ and $S$ are subdivided in the same way in the next step of the hierarchical algorithm.

5. Approaches to systems remodularization
In this section we will discuss some approaches to systems remodularization which are presented in the literature. Most of the papers on this subject are experience reports in which a specific system in a specific environment has been worked upon. Often tools have been built or customized to support the experiments but no general completely automated process has been described.

In [20] a hierarchy (or rather a $(k, 2)$ partite graph) is constructed in which each layer is a view of the system under attention at a particular level of abstraction. The lowest layer consists of pieces of source code. Moving up in the hierarchy, subsystems are composed from elements in a lower
layer in a bottom up manner. This composition is based on structural aspects rather than functional ones. Composition operations are based on software engineering principles like high intra module cohesion and low inter module coupling. At each layer the software structure or interconnection model is represented by a resource-flow graph. The containment relationships between subsystems at different layers form the composition dependency graph. These two types of graphs are combined in the \((k, 2)\) parti- te graph.

A tool called Rigi is used to manipulate the \((k, 2)\) partite graph. The actual composition is to be done by the user, the tool may be of help however in producing alternative clusterings. Measures of composition used are interconnection strength (in terms of the size of interfaces) and clients/suppliers measures. Composition operations provided by the tool are: removal of omnipresent nodes (nodes with a high number of direct clients), composition by interconnection strength, common clients/suppliers, centrality and by name. All these operations are performed if the respective measures lie between specified thresholds.

[5] also presents a hierarchic view of a system. The authors use the module interconnection language NuMIL. In NuMIL a system is a hierarchy of which the interior nodes are called subsystems and the leaf nodes are called modules. A module can be a single function, a collection of functions or even an entire source-code file. The restructuring criteria which are used to extract the subsystems are based on minimizing module coupling and alteration distance, which measures the distance between an altered and an affected module. An algorithm is presented which satisfies these criteria. The modules are represented as nodes in a graph, the edges of this graph represent a path for resource exchange (these resources can be data types, procedures and variables etc.). The algorithm splits up the graph in subgraphs by removing so called articulation points. An articulation point is member of at least two different subgraphs. The articulation points and subgraphs become subsystems.

Another tool which relies on the user to do the intelli- gence work and provides support while doing this is COBOL/SRE which is discussed in [22]. The user is helped to find functionally related pieces of code which are packaged into modules. The main goal of the project was reusable component recovery. For this purpose program segmentation (or slicing) was used but the idea of creating segments to isolate functionally-related groups of statements fits remodularization as well. The tool supports the user in manipulating the segments and produces remodularized COBOL code.

In [26] a tool called Arch is discussed. According to the authors, Arch is a graphical and textual structure chart editor for maintaining large software systems. Arch can be used to group related procedures into modules. A clustering algorithm is provided for this purpose. Another feature of Arch is that it supports ‘maverick analysis’ to find procedures that seem to be assigned to the wrong module. Procedures are found to be related when they share design decisions. Rather than control or data coupling, design coupling is measured. Two procedures are said to have a design link between them when they share assumptions, e.g. about the format and the interpretation of a table. Two procedures can also be put in the same module because they are called by the same procedures. So cross references are treated as features as well (see also [28]). If these features are not enough, Arch allows the user to specify additional features which can be used to express e.g. that procedures use the same algorithm.

In the article requirements on a software similarity measure are presented. One of them demands that similarity is a function of the features common to the two procedures rather than how many possible features are missing from both of them. This is a translation of the ideas behind the Jaccard association coefficient (see section 3.1) to software similarity. The measure used by Arch is not based on features only however, there is an additional term which takes into account if one of the procedures calls the other.

The actual clustering takes place using an hierarchical agglomerative algorithm. The authors have been experimenting with several group similarity measures and report that the single linkage update rule has proved most promising. The tool supports batch clustering, interactive radical clustering where the user is asked for confirmation each time two clusters are combined, and interactive reclustering which uses a previous classification to guide the clustering. Work has been done on automatic tuning of the similarity measure and learning capabilities by use of neural networks (see also [27]).

[11] also group related procedures together. They use data bindings to determine how two procedures are related. Four types of data bindings are distinguished. A potential databinding between procedures \(p\) and \(q\) exists when a variable occurs in the static scope of both \(p\) and \(q\). When both \(p\) and \(q\) actually use (either assign or reference) the variable it becomes a used data binding. An actual data binding exists when \(p\) assigns to a variable which is referenced by \(q\). Finally, when there is a possibility that \(q\) gets control after \(p\) has had control, the binding is called a control flow data binding. The number of control flow data bindings between two procedures is used to compute a similarity measure between each pair of procedures. Several methods for this computation are presented in the article, one of these can be shown to be related to the Jaccard coefficient (see section 3.1). The resulting matrix serves as input for a hierarchical clustering algorithm.

In the article a very intuitive analogy is presented between cluster structures and star systems. Three kinds of structures are distinguished. Planetary systems have several connected subsystems one of which may act as the core of the system. In a black hole system the clustering algorithm
finds one key subsystem which absorbs the rest of the system. Systems that show no tendency to cluster are comparable to gas cloud systems and are probably poorly designed.

In [21] a report is presented on the modularization of a large COBOL program which was partially automated by aid of the tool Software Refinery and the extension RE-
FINE/COBOL. Software Refinery comes with a language, called REFIN which can be used to develop program analysis and transformation tools which are integrated in the tool environment. The entities to be clustered were paragraphs (blocks of statements without their own scope, the flow of control just ‘falls through’). A call graph is generated by the tool after parsing the source. This graph is used to assist the user in selecting the paragraphs to be transferred to another program. The tool then automatically solves all kinds of tricky problems which are inherent to COBOL. For example, data flow analysis is performed to track down the necessary parameters for newly generated call statements.

[6] suggest a restructuring approach which exploits localization and information hiding. By localization the grouping of logically related computational resources into one physical module is meant. Steps in their approach are variable localization, function localization, information hiding and hierarchical packaging, enforcement of information hiding and localization, identification of possible utility packages and abstract data types and finally the re-packaging of improper ADT packages.

In [8] a highly interactive remodularization process is discussed which has been deployed in a Synon/2E environment. In this environment each program ‘belongs to’ a data file. Each data file together with all its programs is called a ‘cluster object’. Cluster objects are the entities to be clustered. After a thorough analysis of the call-relationships between these cluster objects and the development of an ‘application architecture’ which reflected the business perspective, it was determined that there should be seven subsystems. The assignment of cluster objects to modules is done iteratively. The user may assign some cluster objects to appropriate modules using a graphical user interface which visualizes the cluster objects and their interrelationships. At any moment an automated assignment operation can be executed. This operation assigns a cluster object to a module when it is stronger related to the objects in that module than to all other objects together. The process stops when all cluster objects have been assigned to modules.

6. Conclusions

In this article we have presented a general overview of the field of clustering. Clustering methods seem a very good starting point for the remodularization of software. This is because the goal of clustering methods is to group related entities together. An often mentioned problem with clustering methods, namely that many methods impose a structure rather than find ‘natural’ clusters, can be turned into an advantage when applied in restructuring software. We can choose an algorithm which imposes a structure which will satisfy the constraints a good modularization should obey.

Determining the input of the clustering process and specifying criteria for good clusterings will depend heavily on the software environment used. As a software system will continue to evolve after it has been remodularized, incrementally updating a modularization requires special attention. Not much research has been done in this area however.

Another issue which is characteristic to software engineering is that static analysis only reveals which access patterns are possible. Dynamic analysis is needed to determine the weight of the dependencies.

Research sofar has concentrated on experiments and building tools to support these experiments. In this article we provided an overview of both the field of remodularizing software systems and of the general field of cluster analysis. We saw that some existing remodularization techniques use or reinvent cluster algorithms. In our opinion remodularization of software systems could benefit more from the general theory on cluster analysis. We therefore call for more experiments to investigate and exploit the potential synergy of the two fields. Because of the gravity of the problems concerning legacy systems, a lot of systems should be available to serve as the object of such experiments.

In the Resolver project we plan to experiment with cluster algorithms on a real life software system. We aim at fine tuning clustering methods in such a way that the resulting modularization can be said to be more or less ‘object oriented’. The clustering will be used as a starting point to define object classes.

References


